Proceedings of the 17th IEEE International Conference on Automated Software Engineering, 2002, Edinburgh, UK.

Knowledge-Based Synthesis of Numerical Programs for Simulation of Rigid-Body Systems in Physics-Based Animation

Thomas Ellman ellman@cs.vassar.edu
Ryan Deak rydeak@vassar.edu
Jason Fotinatos jafotinatos@vassar.edu

Department of Computer Science
Vassar College
Poughkeepsie, NY 12601

Abstract

Physics-based animation programs are important in a variety of contexts, including education, science and entertainment among others. Manual construction of such programs is expensive, time consuming and prone to error. We have developed a system for automatically synthesizing physics-based animation programs for a significant class of problems: constrained systems of rigid bodies, subject to driving and dissipative forces. Our system includes a graphical interface for specifying a physical scenario, including objects, geometry, dynamical variables and coordinate systems, along with a symbolic interface for specifying forces and constraints operating in the scenario. The entities defined in the graphical interface serve as the underlying vocabulary for specifications constructed in the symbolic interface. We use an algorithmically controlled rewrite system to construct a numerical simulation program that drives a real-time animation of the specified scenario. The algorithm operates by partitioning the constraints and dynamic variables into classes, assigning each class to be implemented in a different component of a general simulation program scheme. Our approach provides many of the benefits of formal deductive methods of program synthesis, while keeping the computational costs of program synthesis more in line with conventional program generator technology. We have successfully tested our system on numerous examples.

1. Introduction

Physics-based animation programs are important in a variety of contexts, including education, science and entertainment among others. For example, in education, animation programs are used to teach the basic principles of physics. In science, physics-based animation programs are used to investigate the behavior of dynamical systems. In entertainment, physics-based animation programs are used in video games involving cars, planes and spaceships, etc. Such programs are usually constructed by hand, in conventional programming languages, such as C++, possibly augmented with a physics-based animation engine toolkit. Unfortunately, manual construction of physics-based animation programs is expensive, time-consuming and highly prone to error.

Our research is aimed at dealing with this problem by applying and extending techniques of Knowledge-Based Software Engineering. We have developed a system for automatically synthesizing physics-based animation programs for a significant class of problems: constrained systems of rigid bodies, subject to driving and dissipative forces. Our system includes a graphical interface (implemented in the MaxScript language of 3D Studio Max®) for specifying a physical scenario, including objects, geometry, dynamical variables and coordinate systems, along with a symbolic interface (implemented in the Mathematica® programming language) for specifying forces and constraints operating in the scenario. The entities defined in the graphical interface serve as the underlying vocabulary for specifications constructed in the symbolic interface. (See Figure 1.) The system automatically generates a C++ program that simulates the behavior of the system specified by the developer, and supplies time-dependent parameters to a rendering engine that generates the animation in real time.

We undertook this research project with a conscious intention of exploiting the well-known tradeoff between the generality of the problem class, and the power of the program synthesis techniques to be developed. We chose to focus on constrained rigid-body mechanics for two reasons. First, we wanted to address a class of problems that is broad enough to include a variety of interesting physical systems. For example, rigid-body systems include vehicles such as cars, planes and sleds; articulated structures such as robots and the human skeleton; along with many other systems commonly appearing in physics-based animation programs. Second, we wanted to address a problem class that would be narrow enough to allow the development and application of powerful, specialized program synthesis techniques. Rigid-body animation programs share a common general framework of interleaved simulation and rendering steps. The framework allows for a large number of variations

regarding the manner in which dynamical variables are formulated and constraints are enforced. The choices among variations are not independent. They depend on each other in fairly complicated ways. Furthermore, the choice among these variations often has a significant impact on the performance of the program – an important issue in real-time animation. Rigid-body animation thus presents a challenging, but ultimately tractable, program synthesis problem.

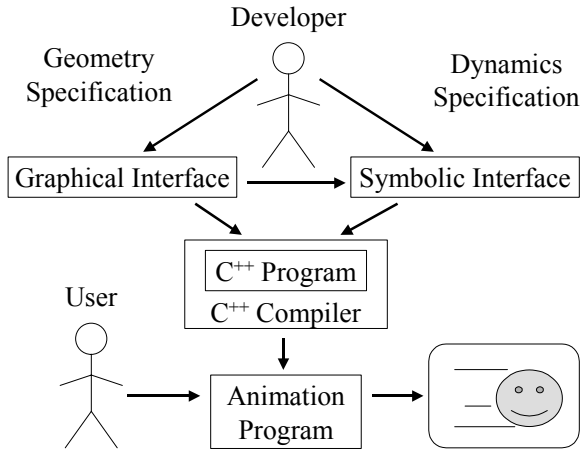

Figure 1. System Architecture

We synthesize rigid-body animation programs using an algorithmically controlled rewrite system. The algorithm operates by partitioning problem constraints and dynamic variables into classes, assigning each class to be implemented in a different component of a general simulation program scheme. The system draws upon a knowledge base of geometry, kinematics, dynamics and numerical methods. Our approach provides many of the benefits of a formal deductive approach to program synthesis, such as the transparency and maintainability that results from use of a declarative specification language and a declarative knowledge base. On the other hand, the computational overhead of program synthesis is considerably lower and more in line with conventional program generator technology. We have successfully tested our system on numerous examples.

2. Dynamics of Rigid-Body Systems

Systems of rigid bodies are the subject of a branch of physics known as "Analytical Dynamics" [1]. In the formalism of Analytical Dynamics, constrained systems of rigid bodies are governed by the Euler-Lagrange equations. (See Figure 2.) These differential equations include expressions involving conservative forces (derived from a potential function, e.g., gravity), nonconservative forces (not derived from a potential function, e.g., dissipative and driving forces) as well as forces derived from constraints. Two types of constraints appear in the equations: Holonomic constraints depend only on the values of dynamical variables, but not on their derivatives. Nonholonomic constraints depend on both the values and the time derivatives of dynamical variables. Each (holonomic or nonholonomic) constraint is associated with a Lagrange multiplier (λ or μ) representing the force that maintains the constraint. The Euler-Lagrange equations may be instantiated in the context of a given rigid-body system by specifying the following things: a set of dynamical variables; the Lagrangian function (kinetic energy minus potential energy); holonomic constraints; nonholonomic constraints and nonconservative forces.

Lagrangian:	$L = T(\mathbf{q},\dot{\mathbf{q}}) - P(\mathbf{q},t)$
Kinetic Energy:	$T(\mathbf{q},\dot{\mathbf{q}})$
Potential Energy:	$P(\mathbf{q},t)$
Dynamical Variables:	$\mathbf{q} = (q_1,...,q_n)$
Time Derivatives:	$\dot{\mathbf{q}} = (\dot{q}_1,...,\dot{q}_n)$
Holonomic Constraints:	$C_i(\mathbf{q},t) = 0$
Nonholonomic Constraints:	$\mathbf{a}_i(\mathbf{q},t) \bullet \dot{\mathbf{q}} + b_i(\mathbf{q},t) = 0$
Nonconservative Forces:	$f_i(\mathbf{q},\dot{\mathbf{q}},t)$
Euler-Lagrange Equations:	$(\mathrm{i} = 1 \ldots \mathrm{n})$

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{q}_i} - \frac{\partial L}{\partial q_i} + \sum_{j=1}^{m} \lambda_j \frac{\partial C_j}{\partial q_i} + \sum_{k=1}^{p} \mu_k a_{ik} = f_i$$

Figure 2. Analytical Dynamics

Consider a single-car roller coaster moving on a circular track with hills and valleys. (See Figures 3 and 4.) The state of the car may be described by three dynamical variables: the angle of revolution of the car around the center of the track, the altitude of the car above the ground, and the pitch angle of the car, which varies as the car goes up and down hills. There are also four additional dynamical variables, one specifying the angle of rotation of each of the four wheels. The (gravitational) potential energy of the car is a simple linear function of height. Treating the car as a point mass, the kinetic energy is $(1/2)mv^2$ where m is the mass of the car, and v is the linear speed of the car in the direction tangent to the car's location on the track. The nonconservative forces are zero, so that the car will neither gain nor lose energy as it moves around the track. The car's motion is governed by several constraints. One (holonomic) constraint asserts that the car's altitude varies as the track goes up and down to keep the car's center of gravity just above the track surface. Another (holonomic) constraint asserts that the car's pitch angle varies to keep the wheels in contact with the surface of the track. A final (nonholonomic)

constraint asserts that the car's wheels rotate so that the relative motion between each tire and the track is zero at the point of contact, i.e., the car does not skid along the surface of the track.

3. Specification of Simulation Programs

3.1 Graphical Specification

In our system, a developer specifies an animation program through a combination of graphical and symbolic interfaces. (See Figure 1.) The graphical interface is implemented in MaxScript, the language of 3D Studio Max. The developer typically begins by defining a tree-structured hierarchy of coordinate systems. A coordinate system hierarchy for the roller coaster animation is shown in Figure 5. The *Root* of the hierarchy is the fixed, global coordinate system for the entire scene. The *CarOrigin* coordinate system is a child of the *Root* system. Its origin is the center of the track. The *Car* coordinate system specifies the location of the car's center of mass. It is defined as a child of the *CarOrigin* system. Coordinate systems locating the wheels, body and canopy of the car are children of the *Car* system. In addition, the *Contact* coordinate system is also a child of the *Car* system. Its origin is the point of contact between the front left tire and the track. After defining a hierarchy of coordinate systems, the developer typically proceeds to define the visual aspects of the animation, including the geometry and light-reflecting properties of visible objects, as well as the locations of lights and cameras. The visible objects, lights and cameras are attached to the leaves of the coordinate system hierarchy.

The coordinate system hierarchy defines a vocabulary in terms of which the developer specifies the dynamical properties of the system to be animated. Each coordinate system is defined in relation to its parent by a translation, a rotation and a scaling operation. The parameters of these transformations (translation vectors, rotation quaternions and scaling factors) are potential variables in the dynamical system being defined by the developer. These transformation parameters may therefore appear in the developer's specification of the Lagrangian, nonconservative forces and constraints governing the behavior of the system. Furthermore the developer may specify the system's constraints in terms of any of the coordinate systems in the hierarchy.

The coordinate system hierarchy also defines the manner in which dynamical variables influence the visual aspects of the scenario. In the final animation program, some of the parent-child transformation parameters will change over time, in response to numerical simulation of the dynamical system. As a result of these changes in parameters, the coordinate systems may change their positions, orientations and/or scale measures. Since the visible objects, lights and cameras are attached to the leaves of the hierarchy, the positions, orientations and sizes of these objects may change as well, resulting in an animation of the physical scenario.

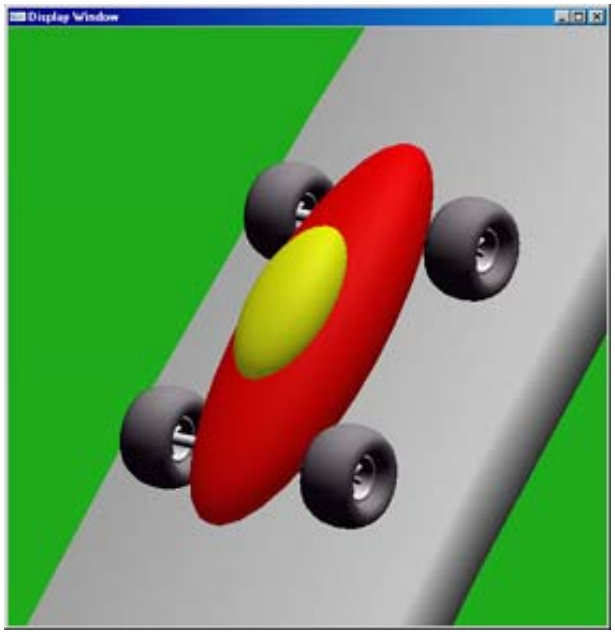

Figure 3. Roller Coaster Car

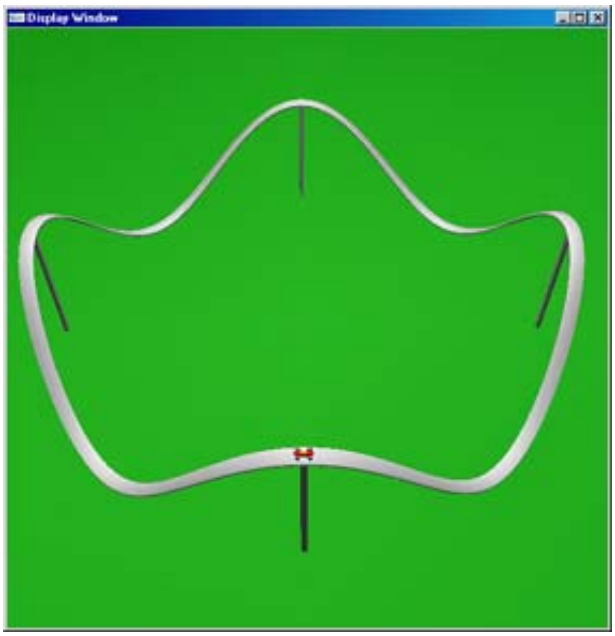

Figure 4. Roller Coaster Track

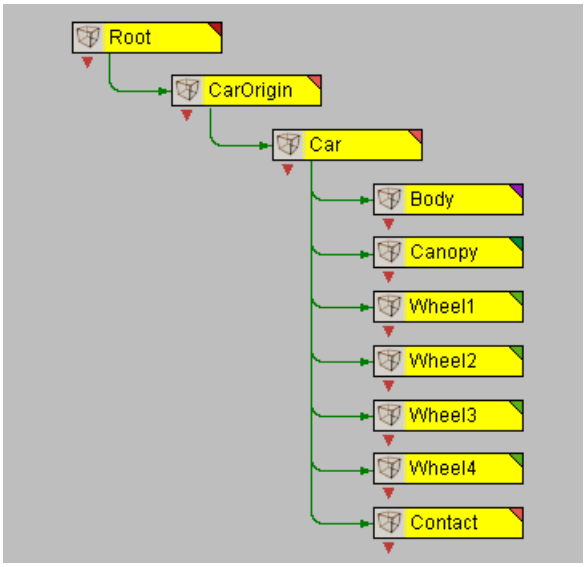

Figure 5. Roller Coaster Coordinate Hierarchy

3.2 Symbolic Specification

The symbolic interface is implemented in the Mathematica programming language. It provides a declarative language in which the developer specifies the dynamics of the system to be animated. An example specification for the roller coaster system is shown in Figure 6. Selected primitives of the specification language are described in Figure 7. The specification uses the following Mathematica notation: The operator *[ ]* indicates function application. A vector is represented as *{x, y, z}*. The *x, y* and *z* components of vector *v* are *v[[1]], v[[2]]* and *v[[3]]*. Definitions are encoded as transformation rules in the form *Lhs -> Rhs*. Each such rule describes how an expression matching *Lhs* may be replaced with the instantiation of *Rhs*. A pattern variable in *Lhs* has an underscore at the end of its name.

The example specification is divided into several sections. In the *Initialization* section, the developer specifies the input parameters to be accepted by the simulation program. These parameters allow the animation to be run multiple times, with different initial states, or different values of forces, masses or other numerical constants appearing in the specification of the dynamics. In the roller coaster specification, there are two input parameters, *InitialAngle* and *InitialOmega.* These represent the initial position of the roller coaster car (as an angle of rotation around the center of the track) and the initial velocity of the car (as an angular velocity). This section also includes *InitializationConstraints*, i.e., equations that relate the input parameters to the initial values of the dynamical variables of the system, thereby giving semantics to the input parameters. In the roller coaster specification, there are two initialization constraint equations. One relates *InitialAngle* to the z-axis Euler angle of the rotation of the *CarOrigin* coordinate system, taken at time zero. The other relates *InitialOmega* to the z-axis component of the angular velocity of the *CarOrigin* coordinate system, also taken at time zero.

In the *DynamicalVariables* section of the specification, the developer provides a list of the system's dynamical variables. Each of these variables represents a parameter to a transformation relating a particular coordinate system to its parent. In the roller coaster example, the specification includes seven dynamical variables. The variable *Rz[CarOrigin]* represents the rotation of the *CarOrigin* coordinate system around the z-axis of its parent, the *Root* system (i.e., the angle of revolution of the car around the center of the track). The variable *Tz[Car]* represents the translation of the *Car* coordinate system along the z-axis of its parent, the *CarOrigin* System (i.e., the vertical motion of the car up and down hills of the track). The variable *Rx[Car]* represents the rotation of the *Car* coordinate system around the x-axis of its parent, the *CarOrigin* system (i.e., the pitch angle of the car). Finally, the variables *Rx[Wheel1]*,…, *Rx[Wheel4]* represent rotations of the car's wheels around the x-axis of the *Car* coordinate system. (Each rotation variable is actually a component of a unit quaternion describing a rotation in three dimensions. A complete quaternion has four components (*Rw,Rx,Ry,Rz*). Whenever the developer includes any of *Rx*, *Ry* or *Rz* in the specification, the system automatically includes *Rw* as well, which is required to maintain normalization of unit quaternions. Thus the roller coaster system actually has a total of thirteen dynamical variables.)

In the *Lagrangian* section, the developer provides a specification of the kinetic and potential energy. The developer need not explicitly define the system's Lagrangian, since the general form of the Lagrangian (as the difference of kinetic and potential energy) is predefined in our system's knowledge base. Instead the developer simply states properties of the objects in the system, which are then used to determine the manner in which kinetic and potential energy are calculated. The *Lagrangian* section includes a list of the mass-bearing bodies along with the mass of each and an indication of whether it should be treated as a *Point* mass (with translational but not rotational kinetic energy) or an extended *Body* mass (with both translational and rotational kinetic energy). General definitions of kinetic energy for point masses and extended bodies are predefined in our system. In particular, the general definition of the kinetic energy of a translating and rotating rigid body is defined in terms of its linear

velocity, angular velocity, mass and inertia tensor. The inertia tensors of a variety of standard shapes (e.g., spheres, toruses, cylinders, etc.) are predefined as well.

Initialization:
```
Inputs -> {InitialAngle,InitalOmega},
InitializationConstraints
 -> {EulerZ[CarOrigin][0] == InitialAngle,
     LocAV[CarOrigin][0][[3]] == InitialOmega }
```

Dynamical Variables:
```
Rz[CarOrigin], Tz[Car], Rx[Car],
Rx[Wheel1], Rx[Wheel2], Rx[Wheel3], Rx[Wheel4]
```

Lagrangian:
```
Masses -> {Car}, Mass[Car] -> 1.0,
MassType[Car] -> Point, PEType -> Unit,
PE[o_][t_] -> Mass[o]*g*AbsTrans[o][t][[3]]
```

Constraints:
```
ForAll[t,AbsTrans[Car][t][[3]] == Altitude[t]],
ForAll[t,Tan[CarPitch[t]] == Slope[t]],
ForAll[t,CPV[t][[2]] == 0],
ForAll[t,Wheel1AV[t][[1]] == Wheel2AV[t][[1]]],
ForAll[t,Wheel1AV[t][[1]] == Wheel3AV[t][[1]]],
ForAll[t,Wheel1AV[t][[1]] == Wheel4AV[t][[1]]]
```

Definitions:
```
Altitude[t_] -> Amplitude*Cos[Frequency*CarRev[t]],
Slope[t_] -> D[Altitude[t],CarRev[t]] / R,
Amplitude -> 12.5, Frequency -> 4, R -> 50.0,
CarRev[t_] -> EulerZ[CarOrigin][t],
CarPitch[t_] -> EulerX[Car][t],
Wheel1AV[t_] -> LocAV[Wheel1][t],
Wheel2AV[t_] -> LocAV[Wheel2][t],
Wheel3AV[t_] -> LocAV[Wheel3][t],
Wheel4AV[t_] -> LocAV[Wheel4][t],
CarLV[t_] -> XFormV[AbsLV[Car][t],Root,Car][t]
WAV[t_] -> XFormAV[Wheel1AV[t],Wheel1,Car][t],
ContactRV[t_] - > RelTrans[Contact,Car][t]
                - RelTrans[Wheel1,Car][t]
CPV[t_ ] -> CarLV[t] + Cross[WAV[t],ContactRV[t]]
```

Figure 6. Roller Coaster Dynamics Specification

The *Lagrangian* section also includes a specification of the system's potential energy function. Two classes of potential energy function are supported. A *Unit* potential is a sum, over all the masses, of the potential energy contribution from each mass. A *Pair* potential is a sum, over all pairs of masses, of an interaction potential for each pair. Together these classes allow a developer to specify most types of potential energy functions encountered in practice, e.g., a uniform gravitational field near the surface of the earth; an N-body Newtonian gravitational potential; and the potential function of a spring governed by Hooke's law, among others. In the roller coaster *Lagrangian* specification, the car is treated as a point mass. Its potential energy is the standard linear function of altitude that describes the uniform gravitational field near the surface of the earth.

The *Constraints* section is a set of temporally quantified equations involving dynamical variables (representing holonomic constraints) or involving both dynamical variables and their derivatives (representing non-holonomic constraints). The *Definitions* section is a syntactic convenience. It allows the user to define functions that are referenced in the *Constraints* section. In the roller coaster specification, there are six constraints. The first (holonomic) constraint asserts that the altitude of the car is equal to the height of the track at the car's current angle of revolution around the center of the track. (The car is on the track.) The second (holonomic) constraint asserts that the tangent of the car's pitch angle is equal to the slope of the track, at the car's current angle of revolution. (The wheels are in contact with the surface of the track.) The third (nonholonomic) asserts that the velocity of the left front wheel is zero at the point at which the wheel makes contact with the track. (The car is not skidding.) In the *Definitions* section, the contact point velocity is expressed as a sum of two contributions: (1) the motion of the contact point due to the car's linear velocity and (2) the motion of the contact point due to the wheel's angular velocity. The second of these two components is a cross product of the wheel's angular velocity and the displacement of the contact point relative to the center of the wheel. These velocities are transformed into the *Car* coordinate system. The no-skid constraint actually asserts that the contact point has zero velocity along the y-axis of the *Car* coordinate system, since that is the direction in which the car is moving. The remaining constraints require the other three wheels to have the same angular velocity as the front left wheel. (Several additional constraints are needed in the roller coaster specification. For each rotating coordinate system, the system automatically adds a constraint requiring the corresponding quaternion to be normalized. Thus the roller coaster is governed by a total of twelve constraints.)

Rules implementing selected primitives in our specification language are shown in Figure 8. These rules use additional Mathematica notation: A rule of the form *Lhs /; Cond := Rhs* asserts that an expression matching *Lhs* and satisfying condition *Cond* can be replaced with the instantiation of *Rhs*. The expression *Dt[f[t],t]* is the total derivative of *f* with respect to *t*. The expression *Grad[f[{x,y,z}],Cartesian[x,y,z]]* is the gradient of *f* in Cartesian coordinates. The expression *E /. R* indicates the application of rule set *R* to expression *E*.

- **Tx[o][t], Ty[o][t], Tz[o][t], Rx[o][t], Ry[o][t], Rz[o][t], Sx[o][t], Sy[o][t], Sz[o][t]**: The translation, rotation or scale of an object o, relative to it's parent, at time t.
- **Masses:** A list of all the mass-bearing objects.
- **MassType[o]:** The mass type of object o: **Point** for point masses and **Body** for extended bodies.
- **PEType:** The type of potential energy function that governs the system: **Unit** or **Pair**.
- **Mass[o], InertiaTensor[o]:** The mass of an object o and its inertia tensor.
- **KE[t], KE[o][t]:** Total kinetic energy and kinetic energy of object o, at time t.
- **PE[t], PE[o][t], PE[a,b][t]:** Total potential energy, potential energy of object o, and interaction potential of objects a and b, at time t.
- **AbsTrans[o][t], AbsRot[o][t], AbsScale[o][t]:** The position, rotation or scale of an object o in the root coordinate system, at time t.
- **RelTrans[a,b][t], RelRot[a,b][t], RelScale[a,b][t]:** The position, rotation or scale of object a relative to object b, at time t.
- **AbsLV[o][t], AbsAV[o][t]:** The linear or angular velocity of object o in the root coordinate system, at time t.
- **LocLV[o][t], LocAV[o][t]:** The linear or angular velocity of object o in its own coordinate system, at time t.
- **EulerX[o][t], EulerY[o][t], EulerZ[o][t]:** The x, y or z Euler angle of the rotation of object o relative to its parent, at time t.
- **XfnP[p,f,g][t], XfnV[v,f,g][t]**, **XfnAV[w,f,g][t], XfnN[n,f,g][t]:** A position, linear velocity, angular velocity or normal vector transformed from coordinate system f to coordinate system g, at time t.
- **LocNormal[s,p]:** The unit vector normal to surface s at relative location p, in the coordinate system of s.
- **AbsNormal[s,p][t]:** The unit vector normal to surface s at absolute location p, in the root coordinate system, at time t.
- **Contains[s,o][t]:** A predicate asserting that object o's origin lies in surface s, at time t.

Figure 7. Selected Specification Language Primitives

First consider the rules implementing the Lagrangian: Potential energy is either a sum over individual masses (Unit potential) or a sum over pairs of masses (Pair potential). An object's potential energy usually depends on its absolute translation. This is computed by applying translation, rotation and scaling transformations along the path from the object to the root of the coordinate system hierarchy. Total kinetic energy is a sum of the kinetic energies of the masses, including translational energy for point masses and both translational and rotational energy for extended bodies. The translational kinetic energy of an object depends on its absolute linear velocity, i.e., the derivative of its absolute translation. Both kinetic and potential energy depend on parameters (*Tx, Ty, Tz, Rx, Ry, Rz, Sx, Sy, Sz*) of transformations in the coordinate system hierarchy, or their derivatives, some of which may be dynamical variables.

Implementing the Lagrangian:

```
Lagrangian[t_] := KE[t] - PE[t]
KE[t_] := Sum[KE[Masses[[m]]][t],{m, NumMasses}]
KE[o_][t_] := TKE[o][t] + RKE[o][t]
TKE[o_][t_] := (1/2) * Mass[o] * (AbsLV[o][t])^2
RKE[o_][t_] /; MassType[o] == Point := 0
RKE[o_][t_] /; MassType[o] == Body
  := (1/2)Mass[o]
     * LocAV[o][t] . InertiaTensor[o] . LocAV[o][t]
PE[t_] /; PEType == Unit
  := Sum[PE[Masses[[m]]][t],{m,NumMasses}]
PE[t_] /; PEType == Pair
  := Sum[PE[Masses[[m1]],Masses[[m2]]][t],
          {m1,NumMasses},{m2,NumMasses}]
AbsLV[o_][t_] := Dt[AbsTrans[o][t], t]
AbsTrans[o_][t_] := RelTrans[o, Root][t]
RelTrans[o_,o_][t_] := ZeroVector
RelTrans[o1_,o2_][t_] /; o1 =!= o2
 := ApplyTrans[RelTrans[Parent[o1], o2][t],
     ApplyRot[RelRot[Parent[o1], o2][t],
       ApplyScale[RelScale[Parent[o1], o2][t],
                                Trans[o1][t]]]]
Trans[o_][t_] := {Tx[o][t], Ty[o][t], Tz[o][t]}
```

Implementing Constraints:

```
Contains[s_, o_][t_]
  := SurfFn[s][XfnP[AbsTrans[o][t],Root,s][t]] == 0
AbsNormal[s_][p_][t_]
  := XfnN[LocNormal[s][XfnP[p,Root,s][t]],s,Root][t]
LocNormal[s_][{a_, b_, c_}]
  := Normalize[Grad[SurfFn[s][{x, y, z}],
                 Cartesian[x, y, z]]  /. {x->a, y->b, c->z}]
XfnP[p_,f_,g_][t_]
 := XfnPDown[XfnPUp[p,f,LCA[f,g]][t], LCA[f,g],g][t]
XfnPUp[p_,f_,lca_][t_]
 := ApplyTrans[RelTrans[f,lca][t],
              ApplyRot[RelRot[f,lca][t],
                       ApplyScale[RelScale[f,lca][t],p]]]
XfnPDown[p_,lca_,g_][t_]
 := ApplyScale[RelScaleInv[g,lca][t],
            ApplyRot[RelRotInv[g,lca][t],
                     ApplyTrans[RelTransInv[g,lca][t],p]]]
```

Figure 8. Rules Implementing Selected Primitives

Now consider the rules for implementing constraints. One rule implements a predicate asserting that a point lies in an implicit surface, i.e., a surface defined by a function of the form *f(x,y,z)=0*. Another rule describes how to compute the unit vector normal to a surface at a point, by evaluating the gradient of the defining function *f*, and then normalizing the result. These rules depend on techniques for transforming vectors between coordinate systems. A position vector *p* is transformed from coordinate system *f* to coordinate system *g* by finding the least common ancestor *a* of *f* and *g*; transforming *p* from *f* to *a*; and then transforming the result from *a* to *g*. These primitives can be used together to assert that one surface is tangent to another surface. Tangency is useful in defining constraints asserting that one object slides or rolls across another.

4. Rigid-Body Simulation Programs

Real-time physics-based animation programs typically operate by repeating the following two steps: (1) Numerically simulate the behavior of the rigid-body system over a short period of time; (2) Render an image of the current state of the system. In order for this process to operate in real time, the simulation step must be fast enough to be executed many times per second. This has important implications for the program synthesis process.

The general scheme of a program for simulating a constrained rigid-body system is shown in Figure 9. The main program begins by solving the initial position and velocity constraints for the initial values of the position and velocity variables. The main loop repeatedly calls a numerical integration routine to integrate a set of dynamical variables over a short time interval. The integration routine solves a set of differential equations that are first-order in some variables and second-order in other variables. It takes a system derivative function as a parameter and calls this derivative function repeatedly during the numerical integration process. After each integration step, the main loop calls a stabilization routine that adjusts the values of some dynamical variables in order to maintain numerical stability. After stabilization, it updates the values of other dynamical variables by solving algebraic constraints.

In our simulation program scheme, the dynamical variables have been partitioned into three groups: P_0, P_1 and P_2. Each variable group is handled in a different component of the simulation program scheme. Likewise, the constraints have been partitioned into three groups: C_0, C_1 and C_2. Each constraint is enforced in a different component of the simulation program scheme. Variables in the group P_0 (zeroth-order variables) are updated at the end of each simulation step, by solving constraints in the group C_0. Variables in the groups P_1 and P_2 are updated by numerical integration. Groups P_1 (first-order variables) and P_2 (second-order variables) differ in the way they appear in the integration process. The variables of integration include P_1 and P_2 along with the derivatives P_2' of variables in P_2, but not the derivatives of variables in P_1. The system derivative function takes the variables of integration (P_1, P_2 and P_2') as input and computes their derivatives (P_1', P_2' and P_2''). The derivatives in P_1' (velocities of variables in group P_1) are computed by solving the derivatives C_1' of constraints in group C_1. The derivatives in P_2' (velocities of variables in group P_2) are obtained from the input to the system derivative function. The derivatives in P_2'' (accelerations of variables in group P_2) are computed by solving the Euler-Lagrange equations, and the second derivatives C_2'' of constraints in group C_2.

EL…….. Euler-Lagrange equations.
CIP, CIV……. Initial position / velocity constraints.
P_0, P_1, P_2 …… 0^{th}, 1^{st} and 2^{nd} order variables.
P_1', P_2', P_2''..… Derivatives of variables in P_1 and P_2.
C_0, C_1, C_2…… 0^{th}, 1^{st} and 2^{nd} order constraints.
C_1', C_2', C_2''..… Derivatives of constraints in C_1 and C_2.

Derivative(P_1,P_2,P_2',t):
a. Solve equations in EL and constraints in C_2'' for variables in P_2''.
b. Solve constraints in C_1' for variables in P_1'.
c. Return (P_1',P_2',P_2'').

Stabilize(P_1,P_2,P_2'):
a. Adjust variables in P_1, P_2 and P_2' to satisfy constraints in C_1, C_2 and C_2'.
b. Return (P_1,P_2,P_2').

Step(P_0,P_1,P_2,P_2',t,dT):
a. (P_1,P_2,P_2') = Integrate(P_1,P_2,P_2',t,dT,Derivative).
b. (P_1,P_2,P_2') = Stabilize(P_1,P_2,P_2').
c. Solve constraints in C_0 for variables in P_0.
d. Return (P_0,P_1,P_2,P_2').

Main Program:
1. Solve constraints CIP for variables in P_0, P_1 and P_2.
2. Solve constraints CIV for variables in P_2'.
3. Render(P_0,P_1,P_2).
4. Let t = 0.
5. Repeat:
    a. (P_0,P_1,P_2,P_2') = Step(P_0,P_1,P_2,P_2',t,dT).
    b. Let t = t + dT.
    c. Render(P_0,P_1,P_2).

Figure 9. Simulation Program Scheme

The performance of the simulation program depends on the manner in which the dynamical variables and constraints are partitioned into the groups described above. In order to see why, consider the following: In the context of real-time animation, the system derivative function must be evaluated many times per second. When a rigid-body system includes more than a few variables, the system derivative usually cannot be solved analytically. Instead, it must be computed by solving a system of linear equations whose size is equal to the number of second-order dynamical variables $|P_2|$ plus the number of second-order constraints $|C_2|$. Solving the linear system is an $O(n^3)$ process, where $n = |P_2|+|C_2|$ is the size of the linear system. We should therefore expect the performance of the simulation process to degrade rapidly if *n* becomes too large. On the other hand, performance will improve to the extent that we can formulate the simulation program in a way that places more variables in groups P_0 and P_1 (rather than P_2) and more constraints in groups C_0 and C_1 (rather than C_2), i.e., when constraints and variables are handled outside of the integration process, or when they are represented in terms of their first, rather than second derivatives.

Our simulation programs incorporate several routines from the Numerical Recipes [2] library, including a Runge-Kutta routine for integration of differential equations; an LU decomposition routine for solving systems of linear algebraic equations, and a Newton-Raphson routine for solving systems of nonlinear algebraic equations. We also use a modified version of the Newton-Raphson routine to solve under-constrained systems of nonlinear algebraic equations, in the program component that maintains the numerical stability of the simulation process. Our approach to simulation of rigid-body systems is based on numerical techniques described in [3]; however, we use a Runge-Kutta method, rather than a predictor-corrector method, for carrying out the main integration step. We also use a different method of maintaining numerical stability.

5. Synthesis of Simulation Programs

A number of questions must be answered in order to synthesize programs instantiating our simulation program scheme:

- **Handling of dynamical variables:** For each dynamical variable, can it be placed in group P_0 and updated outside the numerical integration process? If not, can it be placed in group P_1 so that the variable appears in the integration, but its derivative does not? Or must it be placed in group P_2 so that the variable and its first derivative both appear in the integration?
- **Handling of constraints:** For each constraint, can it be placed in group C_0 and enforced outside of the integration process in constraining variables in group P_0? If not, can it be placed in group C_1 and enforced inside the integration process by constraining first derivatives of variables in group P_1? Or must it be placed in group C_2, enforced during the integration process by constraining second derivatives of variables in group P_2, and appear in the Euler-Lagrange equations?
- **Solution methods:** What computational method should be used to solve each equation or enforce each constraint? Analytic solution? Numeric solution? If numeric, what numerical method should be used?
- **Decomposition:** Can the numerical integration or any of the numerical equation or constraint solutions be decomposed into components that can be solved independently of each other? (We have addressed only the first three groups of questions in our work to date. We plan to investigate decomposition techniques in our future work.)

Our program synthesis algorithm is outlined in Figure 10. The algorithm begins by constructing an initial version of the Lagrangian, based on the input specification. The main part of the algorithm concerns partitioning of dynamical variables into classes P_0, P_1 and P_2 and partitioning constraints into classes C_0, C_1 and C_2, based on the analytic solvability of groups of constraints and their derivatives. Along the way, the constraints are rewritten into equivalent forms referencing smaller sets of variables. The Euler-Lagrange equations are formulated to include a subset of the original constraints and a subset of the original variables.

The behavior of our program synthesis algorithm can be illustrated by considering the roller coaster example. In the initial formulation, the roller coaster specification has thirteen dynamical variables and twelve constraints. These sets of variables and constraints are classified in the following way: In step (3b), the car altitude constraint is solved analytically to obtain a value of the dynamical variable *Tz[Car]* as a function of the car's angle of revolution around the center of the track. The altitude constraint is placed in the constraint group C_0, and the variable *Tz[Car]* is placed in the variable group P_0. The variable *Tz[Car]* is eliminated from the Lagrangian. On the other hand, the car pitch angle constraint (on variables *Rw[Car]* and *Rx[Car]*) and the associated quaternion normalization constraint do not yield a unique analytic solution for the pitch angle. Nevertheless, in step (3c), the first derivative of these constraints is found to have a unique analytic solution. The pitch angle and quaternion normalization constraints are placed in constraint group

C_1 and the variables *Rw[Car]* and *Rx[Car]* are placed in variable group P_1. Since the Lagrangian does not depend on the derivative of the pitch angle, it is not further revised in this step. In a similar manner, all of the wheel rotation constraints, and (the derivatives of) their associated quaternion normalization constraints, are solved analytically. All of the wheel rotation constraints are placed in constraint group C_1, and all of the wheel rotation variables are placed in variable group P_1. Once again, since the Lagrangian does not depend on the derivatives of the wheel rotation variables, it is not further revised at this point. Finally, in step (3d), the variables *Rw[CarOrigin]* and *Rz[CarOrigin]* are placed in the group P_2, and the quaternion normalization constraint associated with these variables is placed in constraint group C_2. A naïve implementation of the roller coaster would have placed all 13 variables in group P_2 and all 12 constraints in group C_2, resulting in a system derivative function that solves a linear system of 25 equations and unknowns. After classifying the variables and constraints as described above, computation of the system derivative requires solving several linear systems, the largest of which has 3 equations and unknowns, and each of which is small enough to be solved analytically.

The final step in program synthesis is to generate C++ code implementing each component of the simulation program scheme. Generation of code is carried out by instantiating several predefined schemata. Each schema describes a function that solves one or more sets of equations. The instantiation process begins by determining whether an analytic solution is available. If so, the analytic solution is incorporated directly into the function schema using a procedure that converts algebraic Mathematica expressions into equivalent C++ strings. If a numerical solution method is required, the system chooses an appropriate method (e.g., LU decomposition for linear equations or Newton-Raphson for nonlinear equations) and generates code that calculates the data used in the selected method (e.g., a matrix for LU decomposition or an array of residual values for Newton-Raphson), again by converting algebraic Mathematica expressions to C++ strings for each matrix or array entry. Many large and complicated sub-expressions appear in multiple locations in the resulting code. For this reason, we carry out an optimization step that identifies repeated sub-expressions, stores them in temporary variables, and arranges for their values to be referenced whenever they are needed.

Our system is implemented in Mathematica, using its facilities for algorithmically manipulating and applying transformation rules. Most of the implementation consists of purely declarative rules implementing knowledge of kinematics (reasoning about coordinate systems, coordinate transformations, velocities, angular velocities, kinetic energy and potential energy, and for constructing the Lagrangian function); geometry (formulating constraints involving surfaces, normal vectors and contact between surfaces); dynamics (constructing the Euler-Lagrange equations) and numerical methods (selecting methods for solving equations and constraints). The algorithmic portion of our system includes the procedure for classifying variables and constraints, and the procedures for generating C++ code, both of which use some imperative features of the Mathematica language. Our system also uses Mathematica's tools for symbolic differentiation, analytic solution of algebraic equations, and simplification of algebraic expressions.

1. Let V be the set of all dynamical variables and C be the set of all constraints.
2. Construct the system Lagrangian L by expanding the definitions of kinetic and potential energy.
3. Partition V into classes P_0, P_1 and P_2 and partition C into classes C_0, C_1 and C_2:
    a. Partition C into holonomic constraints H and nonholonomic constraints N, based on whether they depend on derivatives of dynamical variables.
    b. Find the largest $P_0 \subseteq V$ and $C_0 \subseteq H$ such that C_0 can be solved analytically for P_0 in terms of variables in $V\text{-}P_0$. Let $V=V\text{-}P_0$. Let V' be the derivatives of variables in V. Let $H=H\text{-}C_0$. Let H' be the derivatives of constraints in H. Rewrite H' and N to be free of variables in P_0 and their derivatives.
    c. Find the largest $P_1 \subseteq V$ and $C_1' \subseteq H' \cup N$ such that C_1' can be solved analytically for P_1' in terms of variables in $V \cup (V'\text{-}P_1')$. Let $P_2=V\text{-}P_1$. Let $C_2'=(H' \cup N)\text{-}C_1'$. Let C_2'' be the derivatives of the constraints in C_2'. Rewrite C_2' and C_2'' to be free of variables in P_1' and their derivatives.
4. Construct the Euler-Lagrange equations EL from the Lagrangian L and constraints C_2', all of which are nonholonomic, since the holonomic ones were differentiated with respect to time. Rewrite EL to be free of variables in P_0 and P_1' and their derivatives, using formulae obtained from solving constraints in C_0 and C_1' above.
5. Generate code for each of the program components in the simulation program scheme: For each component that solves a system of equations, use an analytic solution, if possible. Otherwise, if the equations are linear, use a numerical method for solving linear equations. Otherwise use a numerical routine for solving nonlinear equations.

Figure 10. Program Synthesis Algorithm

6. Experimental Results

Our system has been successfully tested on roughly a dozen qualitatively distinct example problems. A summary of these results is shown in Figure 11. In each of these example programs, the developer carried out the following steps: (1) Enter the graphical and symbolic components of the specification; (2) Execute the program synthesis algorithm; (3) Compile the generated C++ code; (4) Execute the resulting animation program. The program synthesis phase takes a period of time ranging from about 30 seconds on the simplest problems to about 25 minutes on the most complicated. We have not made any great effort to optimize the speed of our program synthesis system. We therefore expect that these times may be considerably reduced in future implementations of our system.

- **Pendulum**: Demonstrates basic system operation.
- **Double Pendulum**: One pendulum hangs off another. Demonstrates coordinate system hierarchy.
- **Three Body Planetary System**: Demonstrates use of a Newtonian gravitational potential.
- **Two Spring-Coupled Pendula**: Two pendula are linked by a spring. Demonstrates use of a potential describing Hooke's law.
- **Two Rigidly Linked Pendula**: Two pendula are linked by a rigid rod. Demonstrates handling of constraint systems forming a graph, not a tree.
- **Pendula on Spinning Wheel**: Four pendula are attached to wheel spinning with uniform angular velocity. Demonstrates use of a time-dependent holonomic constraint.
- **Nested Rolling Toruses**: One torus rolls along the inner circumference of a second torus, which rolls along the inner circumference of a third torus. Demonstrates nonholonomic constraints.
- **Weighted Ball Rolling on Plane**: Ball with off-center weight rolls erratically across a plane. Demonstrates rotation in three dimensions.
- **Torus Rolling on Plane**: Tilted torus rolls and spins on plane. Demonstrates use of a dummy object to track a point of contact between two surfaces.
- **Ball Rolling Inside Torus**: A ball rolls on the interior surface of a torus. Demonstrates rolling contact between two implicit surfaces.
- **Single-Car Roller Coaster**: Demonstrates partitioning of variables and constraints.
- **Multi-Car Sliding Coaster**: Five cars and no wheels. Constraints enforce fixed distances between cars.

Figure 11. Summary of Experimental Results

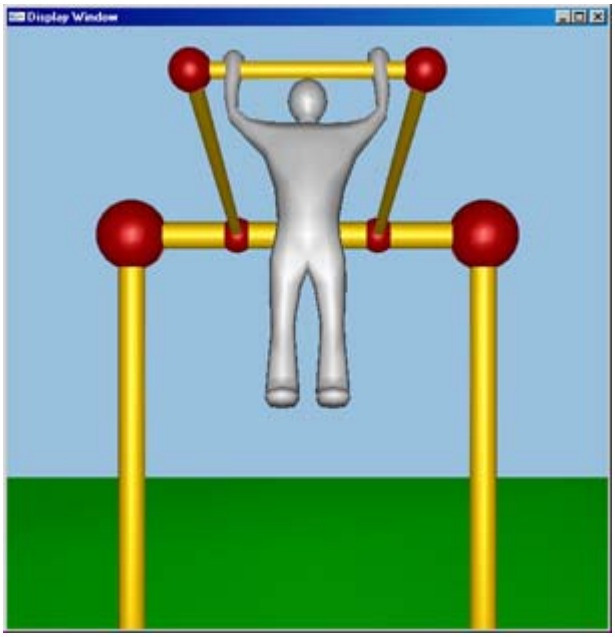

Figure 12. Acrobat on Trapeze

Figure 13. Dancing Snowman

The reader may come away with the impression that the example programs described in Figure 11 are mere exercises in basic mechanics. This is partly true; however, our most dramatic programs result from exploiting the manner in which the dynamical variables of simulation are linked to the visual aspects of a physical scenario. Recall that the developer may attach arbitrary visual objects (surfaces, lights and cameras) to the leaves of the

coordinate system hierarchy defined in the graphical interface. As the simulation unfolds over time, the positions, sizes and orientations of these objects may change as well. The user may view the scene from the point of view of any of the cameras. Some of the resulting effects are illustrated in Figures 12 and 13. With the right choice of geometry, lights and cameras, the double pendulum becomes an "Acrobat on Trapeze", and the ball with off-center weight rolling erratically on a plane becomes a "Dancing Snowman".

7. Related Work

A number of other investigators have also developed automated program synthesis techniques for scientific and numerical computation. The Synapse, Agnes and Ctadel systems are similar to the one we have developed in our work. These systems synthesize programs for solving certain types of partial differential equations (PDEs). Synapse [4] constructs finite element codes, using a knowledge base of transformation rules implemented in Mathematica. Agnes [5] constructs numerical codes by matching input equations to templates, in order to choose an appropriate solution method. Ctadel [6] generates code that runs on sequential, vector and shared virtual and distributed memory architectures. The Sigma, Amphion and AutoBayes systems are also similar to the one we have developed. In Sigma [7], scientific computation problems are specified in terms of a data-flow model, which is executed by an interpreter. In Amphion [8], problem specifications are represented in first-order logic. Amphion uses a deductive method to synthesize a numerical program that meets the specification. AutoBayes [9] generates data analysis programs from declarative descriptions of problem variables and probability distributions. It uses schema-guided deductive synthesis, augmented by symbolic-algebraic computation techniques. Nevertheless, despite these similarities, none of these systems was designed to handle, or is capable of handling, rigid-body simulation problems.

Methods of automating the synthesis of planning and scheduling programs are reported in [10] and [11]. These techniques operate, in part, by assigning each problem constraint to be enforced in an appropriate part of the program being generated. Despite this surface similarity, the applications are so different (symbolic search versus numerical simulation) that the respective program synthesis techniques do not appear to be transferable from either type of application to the other.

The first author's previous work developed deductive methods of synthesizing numerical simulation programs for specifications composed of algebraic and differential equations, with a focus on engineering design applications [12]. In that work, the algebraic and differential equations were fairly simple. The complexity of program synthesis resulted from the variety of logical forms of specifications and the corresponding variety of program architectures for combining numerical codes for integration of differential equations and finding of roots of algebraic equations. In the present work, focusing on synthesis of simulators for rigid-body systems, the architecture for combining numerical codes is relatively fixed. On the other hand, the algebraic and differential equations themselves are much more complex. Furthermore, the synthesis process requires the use of specialized symbolic computation techniques for solving algebraic equations, differentiating constraints and simplifying expressions, which are more efficiently carried out by specialized algorithms, as implemented in a system like Mathematica, rather than a general theorem-proving mechanism. Finally, in the present work, we have addressed the problem of finding an efficient implementation, rather than merely a correct implementation. Standard methods of deductive synthesis are not capable of distinguishing between two correct implementations of differing efficiency. These differences motivated our effort to develop the specialized methods of manipulating the equations and assigning them to components of the simulator architecture, as described above.

Meta-Amphion [13] is a program synthesis system that generates other, domain-specific, program synthesis systems. The domain specific systems incorporate specialized decision procedures into formal deductive methods of program synthesis. In our work, specialized symbolic algebra procedures are an important component of our system's capabilities. It would be interesting to investigate whether a system like Meta-Amphion could synthesize a domain-specific program synthesis system for constructing rigid-body simulation programs, such as the one we have presented here.

8. Contributions

Our research is a contribution to the field of Automated Software Engineering in several different respects. To begin with, real-time 3D animation is becoming a progressively more important part of the software industry. It is now commonly used in computer and video games, as well as in many educational and scientific application programs. As network and processor speeds rise, we expect real-time 3D animation to commonly appear in other contexts, such as web pages and user interfaces to application programs. We have automated the synthesis of an important class of real-time 3D animation programs, i.e., those involving constrained systems of rigid bodies. Our research is therefore a

contribution toward automating a portion of the software engineering problem, the importance of which will grow in the coming years. In addition, our classification-based approach to program synthesis may be applicable to other kinds of software, in which equations and constraints are assigned to be handled in various components of a relatively fixed program architecture. For example, we believe it would be useful in the context of engineering design problems that use a software architecture combining numerical simulation and optimization codes, as described in [14]. Finally, our approach illustrates a methodology that exploits the tradeoff between the power and the generality of a program synthesis system. Our system is a compromise between deductive methods of program synthesis and conventional parameter-driven program generators. On the one hand, we have sacrificed the generality of formal deductive synthesis. In return, we have obtained a system in which the computational costs of program synthesis are more in line with conventional program generator technology. On the other hand, our system uses a declarative specification language and a declarative knowledge base, which provide more maintainability and transparency than are typically seen in conventional program generators. We expect that approaches to program synthesis manifesting this sort of compromise would be useful in other areas of Automated Software Engineering as well.

9. Acknowledgements

The research reported in this paper was supported by Vassar College through a faculty research grant and the Undergraduate Research Summer Institute. The anonymous referees provided thoughtful comments that the authors found quite helpful in preparing the final version of this paper.

10. References

[1] H. Baruh, "Analytical Dynamics", WCB/McGraw-Hill, 1999.

[2] W. Press, W. Vetterling, S. Teukolsky, and B. Flannery, "Numerical Recipes", Cambridge University Press, New York, NY, 1986.

[3] E. Haug, "Computer-Aided Kinematics and Dynamics of Mechanical Systems", Allyn & Bacon, Boston, MA, 1989.

[4] E. Kant, "Synthesis of Mathematical Modeling Software", IEEE Software, 10, 3, 1993.

[5] A. Kowalski and R. Peskin, "Anatomy of AGNES: An Automatic Generator of Numerical Equation Solutions", in E. N. Houstis, Ed., Intelligent Mathematical Software Systems, Elsevier Science Publishers, New York, NY, 1990.

[6] R. van Engelen, L. Wolters, and G. Cats, "Tomorrow's Weather Forecast: Automatic Code Generation for Atmospheric Modeling", IEEE Computational Science & Engineering, 4, 3, 1997.

[7] R. Keller, M. Rimon, and A. Das, "A Knowledge-Based Prototyping Environment for Construction of Scientific Modeling Software", Automated Software Engineering, 1, 1, 1994.

[8] M. Lowry, A. Philpot, T. Pressberger, and I. Underwood, "A Formal Approach to Domain-Oriented Software Design Environments", Proceedings of the Ninth Knowledge-Based Software Engineering Conference, Monterey, CA, 1994

[9] B. Fisher and J. Schumann, "AutoBayes: A System for Generating Data Analysis Programs from Statistical Models", Journal of Functional Programming, 2002, in press.

[10] B. Srivastava and S. Kambhampati, "Synthesizing Customized Planners from Specifications", Journal of Artificial Intelligence Research, 8, 1998.

[11] L. Blaine, L. Gilham, J. Liu, D. Smith, and S. Westfold, "Planware -- Domain-Specific Synthesis of High-Performance Schedulers", Proceedings of the Thirteenth Automated Software Engineering Conference, IEEE Computer Society Press, Los Alamitos, CA, 1998.

[12] T. Ellman and T. Murata, "Deductive Synthesis of Numerical Simulation Programs from Networks of Algebraic and Ordinary Differential Equations", Automated Software Engineering, 5, 3, 1998.

[13] M. Lowry and J. Van Baalen, "Meta-Amphion: Synthesis of Efficient Domain-Specific Program Synthesis Systems", Automated Software Engineering, 4, 2, 1997.

[14] T. Ellman, J. Keane, A. Banerjee and G. Armhold, "A Transformation System for Interactive Reformulation of Design Optimization Strategies", Research in Engineering Design, 10, 1, 1998.